1 Getting started with ASP.NET Core

This chapter covers

- What is ASP.NET Core?
- Things you can build with ASP.NET Core
- How ASP.NET Core works

Choosing to learn and develop with a new framework is a big investment, so it's important to establish early on whether it's right for you. In this chapter, I provide some background on ASP.NET Core: what it is, how it works, and why you should consider it for building your web applications.

By the end of this chapter, you should have a good overview of the benefits of ASP.NET Core, the role of .NET 7, and the basic mechanics of how ASP.NET Core works. So without further ado, let's dive in!

1.1 What is ASP.NET Core?

ASP.NET Core is a cross-platform, open-source application framework that you can use to build dynamic web applications quickly. You can use ASP.NET Core to build server-rendered web applications, backend server applications, HTTP APIs that can

be consumed by mobile applications, and much more. ASP.NET Core runs on .NET 7, which is the latest version of .NET Core—a high-performance, cross-platform, open-source runtime.

ASP.NET Core provides structure, helper functions, and a framework for building applications, which saves you from having to write a lot of this code yourself. Then the ASP.NET Core framework code calls in to your handlers, which in turn call methods in your application's business logic, as shown in figure 1.1. This business logic is the core of your application. You can interact with other services here, such as databases or remote APIs, but your business logic typically doesn't depend *directly* on ASP.NET Core.

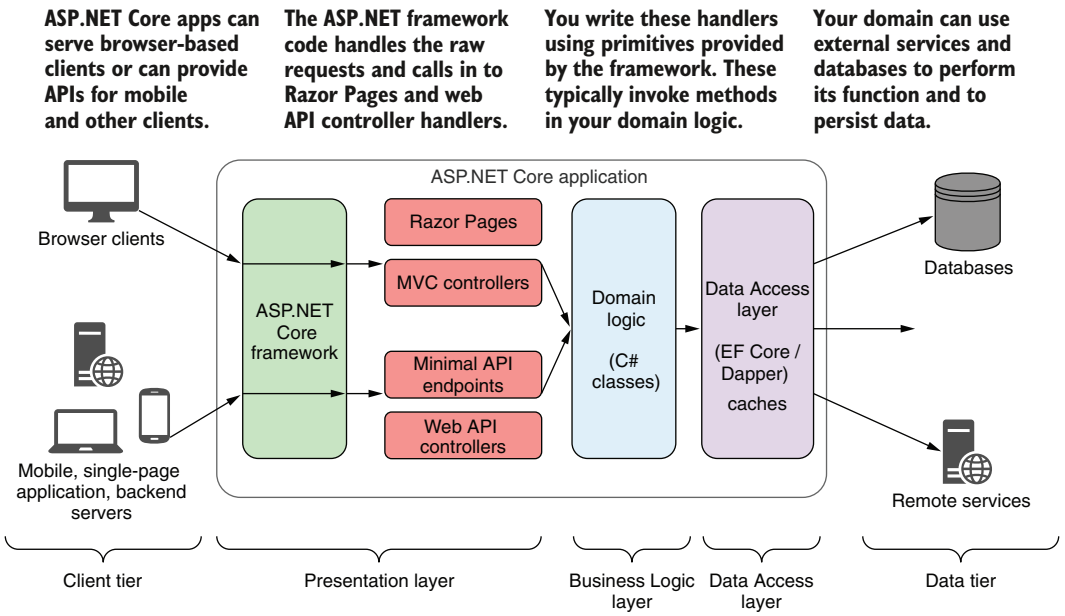

Figure 1.1 A typical ASP.NET Core application consists of several layers. The ASP.NET Core framework code handles requests from a client, dealing with the complex networking code. Then the framework calls in to handlers (Razor Pages and Web API controllers, for example) that you write using primitives provided by the framework. Finally, these handlers call in to your application's domain logic—typically, C# classes and objects without any dependencies that are specific to ASP.NET Core.

1.2 What types of applications can you build?

ASP.NET Core provides a generalized web framework that you can use to build a wide variety of applications. ASP.NET Core includes APIs that support many paradigms:

- *Minimal APIs*—Simple HTTP APIs that can be consumed by mobile applications or browser-based single-page applications.
- *Web APIs*—An alternative approach to building HTTP APIs that adds more structure and features than minimal APIs.
- *gRPC APIs*—Used to build efficient binary APIs for server-to-server communication using the gRPC protocol.

- *Razor Pages*—Used to build page-based server-rendered applications.
- *MVC controllers*—Similar to Razor Pages. Model-View-Controller (MVC) controller applications are for server-based applications but without the page-based paradigm.
- *Blazor WebAssembly*—A browser-based single-page application framework that uses the WebAssembly standard, similar to JavaScript frameworks such as Angular, React, and Vue.
- *Blazor Server*—Used to build stateful applications, rendered on the server, that send UI events and page updates over WebSockets to provide the feel of a client-side single-page application, but with the ease of development of a server-rendered application.

All these paradigms are based on the same building blocks of ASP.NET Core, such as the configuration and logging libraries, and then place extra functionality on top. The best paradigm for your application depends on multiple factors, including your API requirements, the details of existing applications you need to interact with, the details of your customers' browsers and operating environment, and scalability and uptime requirements. You don't need to choose only one of these paradigms; ASP.NET Core can combine multiple paradigms within a single application.

1.3 Choosing ASP.NET Core

I hope that now you have a general grasp of what ASP.NET Core is and the type of applications you can build with it. But one question remains: should you use it? Microsoft recommends that all new .NET web development use ASP.NET Core, but switching to or learning a new web stack is a big ask for any developer or company.

If you're new to .NET development and are considering ASP.NET Core, welcome! Microsoft is pushing ASP.NET Core as an attractive option for web development beginners, but taking .NET cross-platform means that it's competing with many other frameworks on their own turf. ASP.NET Core has many selling points compared with other cross-platform web frameworks:

- It's a modern, high-performance, open-source web framework.
- It uses familiar design patterns and paradigms.
- C# is a great language (but you can use VB.NET or F# if you prefer).
- You can build and run on any platform.

ASP.NET Core is a reimagining of the ASP.NET framework, built with modern software design principles on top of the new .NET platform. Although it's new in one sense, .NET (previously called *.NET Core*) has had widespread production use since 2016 and has drawn significantly from the mature, stable, and reliable .NET Framework, which has been used for more than two decades. You can rest easy knowing that by choosing ASP.NET Core and .NET 7, you're getting a dependable platform as well as a full-featured web framework.

One major selling point of ASP.NET Core and .NET 7 is the ability to develop and run on any platform. Whether you're using a Mac, Windows, or Linux computer, you can run the same ASP.NET Core apps and develop across multiple environments. A wide range of distributions are supported for Linux users: RHEL, Ubuntu, Debian, CentOS, Fedora, and openSUSE, to name a few. ASP.NET Core even runs on the tiny Alpine distribution, for truly compact deployments to containers, so you can be confident that your operating system of choice will be a viable option.

If you're already a .NET developer, the choice of whether to invest in ASP.NET Core for new applications was largely a question of timing. Early versions of .NET Core lacked some features that made it hard to adopt, but that problem no longer exists in the latest versions of .NET. Now Microsoft explicitly advises that all new .NET applications should use .NET 7 (or newer).

Microsoft has pledged to provide bug and security fixes for the older ASP.NET framework, but it won't provide any more feature updates. .NET Framework isn't being removed, so your old applications will continue to work, but you shouldn't use it for new development.

The main benefits of ASP.NET Core over the previous ASP.NET framework are

- Cross-platform development and deployment
- Focus on performance as a feature
- A simplified hosting model
- Regular releases with a shorter release cycle
- Open-source
- Modular features
- More application paradigm options
- The option to package .NET with an app when publishing for standalone deployments

As an existing .NET developer who's moving to ASP.NET Core, your ability to build and deploy cross-platform opens the door to a whole new avenue of applications, such as taking advantage of cheaper Linux virtual machine hosting in the cloud, using Docker containers for repeatable continuous integration, or writing .NET code on your Mac without needing to run a Windows virtual machine. ASP.NET Core, in combination with .NET 7, makes all this possible.

That's not to say that your experience deploying ASP.NET applications to Windows and Internet Information Services (IIS) is wasted. On the contrary, ASP.NET Core uses many of the same concepts as the previous ASP.NET framework, and you can still run your ASP.NET Core applications in IIS, so moving to ASP.NET Core doesn't mean starting from scratch.

1.4 How does ASP.NET Core work?

I've covered the basics of what ASP.NET Core is, what you can use it for, and why you should consider using it. In this section, you'll see how an application built with

ASP.NET Core works, from a user request for a URL to the display of a page in the browser. To get there, first you'll see how an HTTP request works for any web server; then you'll see how ASP.NET Core extends the process to create dynamic web pages.

1.4.1 How does an HTTP web request work?

As you know now, ASP.NET Core is a framework for building web applications that serve data from a server. One of the most common scenarios for web developers is building a web app that you can view in a web browser. Figure 1.2 shows the high-level process you can expect from any web server.

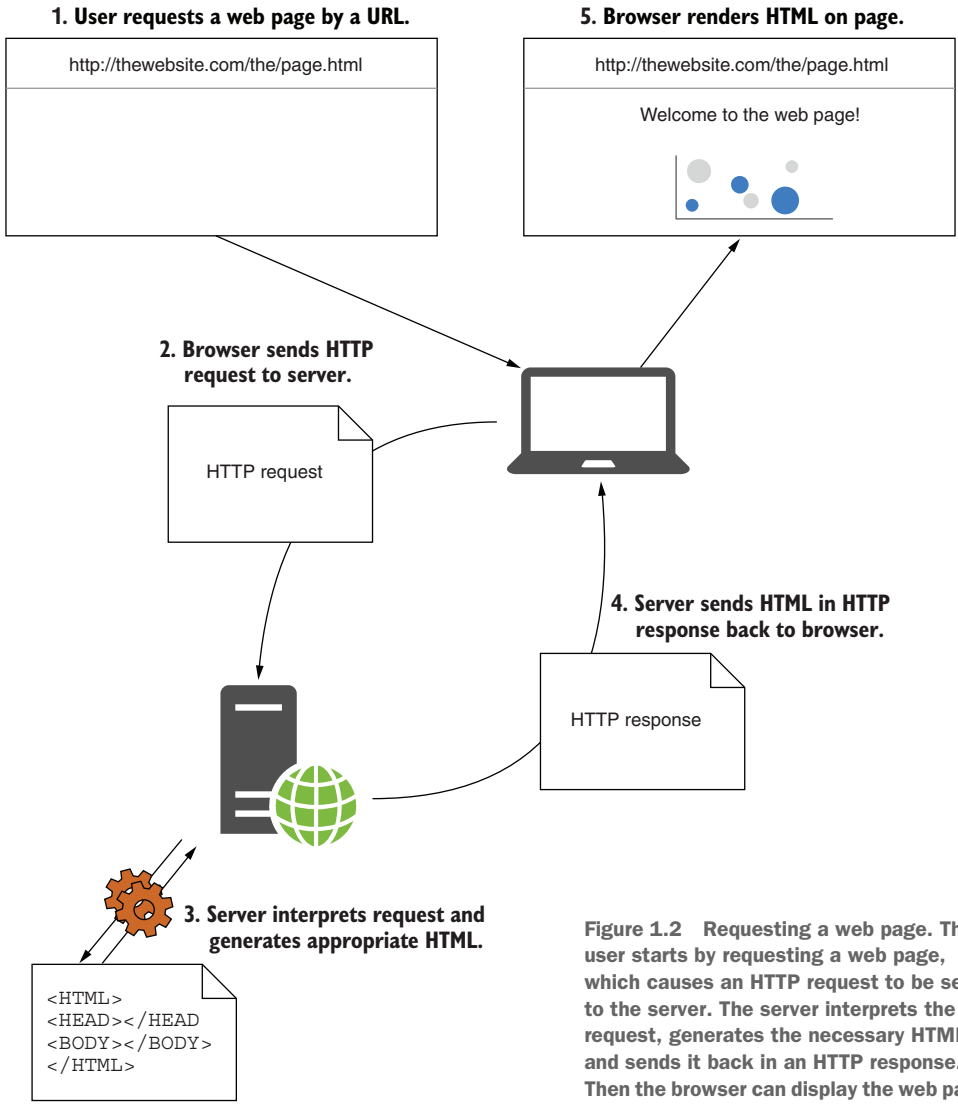

Figure 1.2 Requesting a web page. The user starts by requesting a web page, which causes an HTTP request to be sent to the server. The server interprets the request, generates the necessary HTML, and sends it back in an HTTP response. Then the browser can display the web page.

The process begins when a user navigates to a website or types a URL in their browser. The URL or web address consists of a *hostname* and a *path* to some resource on the web app. Navigating to the address in the browser sends a request from the user's computer to the server on which the web app is hosted, using the HTTP protocol.

DEFINITION The *hostname* of a website uniquely identifies its location on the internet by mapping via the Domain Name Service (DNS) to an IP address. Examples include microsoft.com, www.google.co.uk, and facebook.com.

A brief primer on HTTP

Hypertext Transfer Protocol (HTTP) is the application-level protocol that powers the web. It's a stateless request-response protocol whereby a client machine sends a *request* to a server, which sends a *response* in turn.

Every HTTP request consists of a *verb* indicating the type of the request and a *path* indicating the resource to interact with. A request typically also includes *headers*, which are key-value pairs, and in some cases a *body*, such as the contents of a form, when sending data to the server.

An HTTP response contains a *status code*, indicating whether the request was successful, and optionally *headers* and a *body*.

For a more detailed look at the HTTP protocol itself, as well as more examples, see section 1.3 ("A quick introduction to HTTP") of *Go Web Programming*, by Sau Sheong Chang (Manning, 2016), at http://mng.bz/x4mB. You can also read the raw RFC specification at https://www.rfc-editor.org/rfc/rfc9110.txt if dense text is your thing!

The request passes through the internet, potentially to the other side of the world, until it finally makes its way to the server associated with the given hostname, on which the web app is running. The request is potentially received and rebroadcast at multiple routers along the way, but only when it reaches the server associated with the hostname is the request processed.

When the server receives the request, it processes that request and generates an HTTP response. Depending on the request, this response could be a web page, an image, a JavaScript file, a simple acknowledgment, or practically any other file. For this example, I'll assume that the user has reached the home page of a web app, so the server responds with some HTML. The HTML is added to the HTTP response, which is sent back across the internet to the browser that made the request.

As soon as the user's browser begins receiving the HTTP response, it can start displaying content on the screen, but the HTML page may also reference other pages and links on the server. To display the complete web page instead of a static, colorless, raw HTML file, the browser must repeat the request process, fetching every referenced file. HTML, images, Cascading Style Sheets (CSS) for styling, and JavaScript files for extra behavior are all fetched using exactly the same HTTP request process.

Pretty much all interactions that take place on the internet are a facade over this basic process. A basic web page may require only a few simple requests to render fully, whereas a large modern web page may take hundreds. At this writing, the Amazon .com home page (https://www.amazon.com) makes 410 requests, including requests for 4 CSS files, 12 JavaScript files, and 299 image files!

Now that you have a feel for the process, let's see how ASP.NET Core dynamically generates the response on the server.

1.4.2 How does ASP.NET Core process a request?

When you build a web application with ASP.NET Core, browsers will still be using the same HTTP protocol as before to communicate with your application. ASP.NET Core itself encompasses everything that takes place on the server to handle a request, including verifying that the request is valid, handling login details, and generating HTML.

As with the generic web page example, the request process starts when a user's browser sends an HTTP request to the server, as shown in figure 1.3.

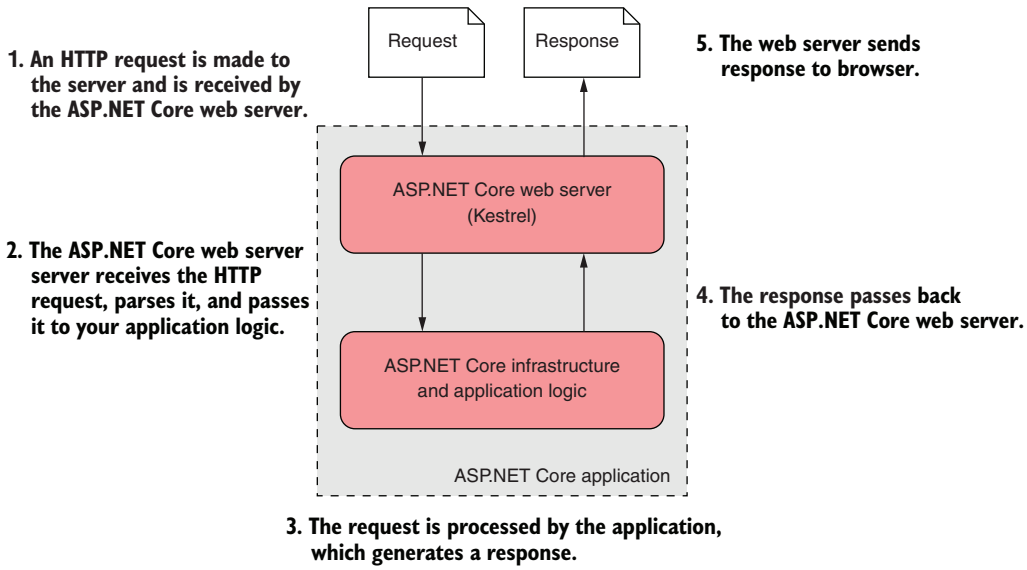

Figure 1.3 How an ASP.NET Core application processes a request. A request is received by the ASP.NET Core application, which runs a self-hosted web server. The web server processes the request and passes it to the body of the application, which generates a response and returns it to the web server. The web server sends this response to the browser.

The request is received from the network by your ASP.NET Core application. Every ASP.NET Core application has a built-in web server—Kestrel, by default—that is responsible for receiving raw requests and constructing an internal representation of the data, an `HttpContext` object, which the rest of the application can use.

Your application can use the details stored in `HttpContext` to generate an appropriate response to the request, which may be to generate some HTML, to return an "access denied" message, or to send an email, all depending on your application's requirements.

When the application finishes processing the request, it returns the response to the web server. The ASP.NET Core web server converts the representation to a raw HTTP response and sends it to the network, which forwards it to the user's browser.

To the user, this process appears to be the same as for the generic HTTP request shown in figure 1.2: the user sent an HTTP request and received an HTTP response. All the differences are server-side, within your application.

You've seen how requests and responses find their way to and from an ASP.NET Core application, but I haven't yet touched on how the response is generated. Throughout this book, we'll look at the components that make up a typical ASP.NET Core application and how they fit together. A lot goes into generating a response in ASP.NET Core, typically within a fraction of a second, but over the course of the book we'll step through an application slowly, covering each of the components in detail.

1.5 What you'll learn in this book

This book takes you on an in-depth tour of the ASP.NET Core framework. To benefit from the book, you should be familiar with C# or a similar object-oriented language. Basic familiarity with web concepts such as HTML and JavaScript will also be beneficial. You'll learn the following:

- How to build HTTP API applications using minimal APIs
- How to create page-based applications with Razor Pages
- Key ASP.NET Core concepts such as model-binding, validation, and routing
- How to generate HTML for web pages by using Razor syntax and Tag Helpers
- How to use features such as dependency injection, configuration, and logging as your applications grow more complex
- How to protect your application by using security best practices

Throughout the book we'll use a variety of examples to learn and explore concepts. The examples are generally small and self-contained so that we can focus on a single feature at a time.

I'll be using Visual Studio for most of the examples in this book, but you'll be able to follow along using your favorite editor or integrated development environment (IDE). Appendix A includes details on setting up your editor or IDE and installing the .NET 7 software development kit (SDK). Even though the examples in this book show Windows tools, everything you see can be achieved equally well on the Linux or Mac platform.

> **TIP** You can install .NET 7 from https://dotnet.microsoft.com/download. Appendix A contains further details on configuring your development environment to work with ASP.NET Core and .NET 7.

In chapter 2, we'll look in greater depth at the types of applications you can create with ASP.NET Core. We'll also explore its advantages over the older ASP.NET and .NET Framework platforms.

Summary

- ASP.NET Core is a cross-platform, open-source, high-performance web framework.
- ASP.NET Core runs on .NET, previously called .NET Core.
- You can use Razor Pages or MVC controllers to build server-rendered, page-based web applications.
- You can use minimal APIs or web APIs to build RESTful or HTTP APIs.
- You can use gRPC to build highly efficient server-to-server RPC applications.
- You can use Blazor WebAssembly to build client-side applications that run in the browser and Blazor Server to build stateful, server-rendered applications that send UI updates via a WebSocket connection.
- Microsoft recommends ASP.NET Core and .NET 7 or later for all new web development over the legacy ASP.NET and .NET Framework platforms.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows you to build responses to a given request dynamically.
- An ASP.NET Core application contains a web server, which serves as the entry point for a request.